APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: DATA TRANSFER MECHANISM

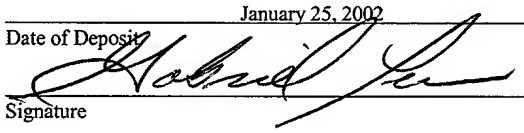
APPLICANT: GILBERT WOLRICH, MARK B. ROSENBLUTH, DEBRA BERNSTEIN AND MATTHEW J. ADILETTA

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL870691429US

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

January 25, 2002
Date of Deposit


Signature

Gabriel Lewis
Typed or Printed Name of Person Signing Certificate

DATA TRANSFER MECHANISM

BACKGROUND

Typical computer processing systems have buses that enable various components to communicate with each other. Bus communication between these components allow transfer of data commonly through a data path. Generally, the datapath interconnects a processing agent, e.g., a central processing unit (CPU) or processor, with other components such as hard disk drives, device adapters, and the like.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a processing system.

FIG. 2 is a detailed block diagram of the processing system of FIG. 1.

FIG. 3 is a flow chart of a read process in the processing system of FIG. 1.

FIG. 4 is a flow chart of a write process in the processing system of FIG. 1.

FIG. 5 is a flow chart of a push operation of the processing system of FIG. 1.

FIG. 6 is a flow chart of a pull operation of the processing system of FIG. 1.

DESCRIPTION

Architecture:

Referring to FIG. 1, a computer processing system 10 includes a parallel, hardware-based multithreaded network processor 12. The hardware-based multithreaded processor 12 is coupled to a memory system or memory resource 14. Memory system 14 includes dynamic random access memory (DRAM) 14a and static random access memory 14b (SRAM). The processing system 10 is especially useful for tasks that can be broken into parallel subtasks or functions. Specifically, the hardware-based multithreaded processor 12 is useful for tasks that are bandwidth oriented rather than latency oriented. The hardware-based multithreaded processor 12 has multiple microengines or programming engines 16 each with multiple hardware controlled threads that are simultaneously active and independently work on a specific task.

The programming engines 16 each maintain program counters in hardware and states associated with the program counters. Effectively, corresponding sets of context or threads can be simultaneously active on each of the programming engines 16 while only one is actually operating at any one time.

In this example, eight programming engines 16 are

illustrated in FIG. 1. Each programming engine 16 has capabilities for processing eight hardware threads or contexts. The eight programming engines 16 operate with shared resources including memory resource 14 and bus interfaces. The hardware-based multithreaded processor 12 includes a dynamic random access memory (DRAM) controller 18a and a static random access memory (SRAM) controller 18b. The DRAM memory 14a and DRAM controller 18a are typically used for processing large volumes of data, e.g., processing of network payloads from network packets. The SRAM memory 14b and SRAM controller 18b are used in a networking implementation for low latency, fast access tasks, e.g., accessing look-up tables, memory for the core processor 20, and the like.

Push buses 26a-26b and pull buses 28a-28b are used to transfer data between the programming engines 16 and the DRAM memory 14a and the SRAM memory 14b. In particular, the push buses 26a-26b are unidirectional buses that move the data from the memory resources 14 to the programming engines 16 whereas the pull buses 28a-28b move data from the programming engines 16 to the memory resources 14.

The eight programming engines 16 access either the DRAM memory 14a or SRAM memory 14b based on characteristics of the data. Thus, low latency, low bandwidth data are stored in and

fetched from SRAM memory 14b, whereas higher bandwidth data for which latency is not as important, are stored in and fetched from DRAM 14a. The programming engines 16 can execute memory reference instructions to either the DRAM controller 18a or SRAM controller 18b.

The hardware-based multithreaded processor 12 also includes a processor core 20 for loading microcode control for other resources of the hardware-based multithreaded processor 12. In this example, the processor core 20 is an XScale™ based architecture.

The processor core 20 performs general purpose computer type functions such as handling protocols, exceptions, and extra support for packet processing where the programming engines 16 pass the packets off for more detailed processing such as in boundary conditions. The processor core 20 has an operating system (not shown). Through the operating system (OS), the processor core 20 can call functions to operate on programming engines 16. The processor core 20 can use any supported OS, in particular a real time OS. For the core processor 20 implemented as an XScale™ architecture, operating systems such as Microsoft NT real-time, VXWorks and µCOS, or a freeware OS available over the Internet can be used.

Advantages of hardware multithreading can be explained by

SRAM or DRAM memory accesses. As an example, an SRAM access requested by a context (e.g., Thread_0), from one of the programming engines 16 will cause the SRAM controller 18b to initiate an access to the SRAM memory 14b. The SRAM controller 18b accesses the SRAM memory 14b, fetches the data from the SRAM memory 14b, and returns data to a requesting programming engine 16.

During an SRAM access, if one of the programming engines 16 had only a single thread that could operate, that programming engine would be dormant until data was returned from the SRAM memory 14b.

By employing hardware context swapping within each of the programming engines 16, the hardware context swapping enables other contexts with unique program counters to execute in that same programming engine. Thus, another thread e.g., Thread_1 can function while the first thread, Thread_0, is awaiting the read data to return. During execution, Thread_1 may access the DRAM memory 14a. While Thread_1 operates on the DRAM unit, and Thread_0 is operating on the SRAM unit, a new thread, e.g., Thread_2 can now operate in the programming engine 16. Thread_2 can operate for a certain amount of time until it needs to access memory or perform some other long latency operation, such as making an access to a bus interface. Therefore, simultaneously,

the processor 12 can have a bus operation, SRAM operation and DRAM operation all being completed or operated upon by one of the programming engines 16 and have one more thread available to process more work.

The hardware context swapping also synchronizes completion of tasks. For example, two threads could hit the shared memory resource, e.g., the SRAM memory 14b. Each one of the separate functional units, e.g., the SRAM controller 18b, and the DRAM controller 18a, when they complete a requested task from one of the programming engine thread or contexts reports back a flag signaling completion of an operation. When the programming engine 16 receives the flag, the programming engine 16 can determine which thread to turn on.

One example of an application for the hardware-based multithreaded processor 12 is as a network processor. As a network processor, the hardware-based multithreaded processor 12 interfaces to network devices such as a Media Access Controller (MAC) device, e.g., a 10/100BaseT Octal MAC 13a or a Gigabit Ethernet device (not shown). In general, as a network processor, the hardware-based multithreaded processor 12 can interface to any type of communication device or interface that receives or sends large amount of data. The computer processing system 10 functioning in a networking application could receive network

packets and process those packets in a parallel manner.

Programming Engine Contexts:

As described above, each of the programming engines 16 supports multi-threaded execution of eight contexts. This allows one thread to start executing just after another thread issues a memory reference and must wait until that reference completes before doing more work. Multi-threaded execution is critical to maintaining efficient hardware execution of the programming engines 16 because memory latency is significant. Multi-threaded execution allows the programming engines 16 to hide memory latency by performing useful independent work across several threads.

Each of the eight contexts of the programming engines 16, to allow for efficient context swapping, has its own register set, program counter, and context specific local registers. Having a copy per context eliminates the need to move context specific information to and from shared memory and programming engine registers for each context swap. Fast context swapping allows a context to perform computations while other contexts wait for input-output (I/O), typically, external memory accesses to complete or for a signal from another context or hardware unit.

For example, the programming engines 16 execute eight

contexts by maintaining eight program counters and eight context relative sets of registers. A number of different types of context relative registers, such as general purpose registers (GPRs), inter-programming agent registers, Static Random Access Memory (SRAM) input transfer registers, Dynamic Random Access Memory (DRAM) input transfer registers, SRAM output transfer registers, DRAM output transfer registers. Local memory registers can also be used.

For example, GPRs are used for general programming purposes. GPRs are read and written exclusively under program control. The GPRs, when used as a source in an instruction, supply operands to an execution datapath (not shown). When used as a destination in an instruction, the GPRs are written with the result of the execution box datapath. The programming engines 16 also include IO transfer registers as discussed above. The IO transfer registers are used for transferring data to and from the programming engines 16 and locations external to the programming engines 16, e.g., the DRAM memory 14a and the SRAM memory 14b etc.

Bus Architecture:

Referring to FIG. 2, the hardware-based multithreaded processor 12 is shown in greater detail. The DRAM memory 14a and

the SRAM memory 14b are connected to the DRAM memory controller 18a and the SRAM memory 18b, respectively. The DRAM controller 18a is coupled to a pull bus arbiter 30a and a push bus arbiter 32a, which are coupled to a programming engines 16a. The SRAM controller 18b is coupled to a pull bus arbiter 30b and a push bus arbiter 32b, which are coupled to a programming engine 16b. Buses 26a-26b and 28a-28b make up the major buses for transferring data between the programming engines 16a-16b and the DRAM memory 14a and the SRAM memory 14b. Any thread from any of the programming engines 16a-16b can access the DRAM controller 18a and the SRAM controller 18a.

In particular, the push buses 26a-26b have multiple sources of memory such as memory controller channels and internal read registers (not shown) which arbitrate via the push arbiters 32a-32b to use the push buses 26a-26b. The destination (e.g., programming engine 16) of any push data transfer recognizes when the data is being "pushed" into it by decoding the Push_ID, which is driven or sent with the push data. The pull buses 28a-28b also have multiple destinations (e.g., writing data to different memory controller channels or writeable internal registers) that arbitrate to use the pull buses 28a-28b. The pull buses 28a-28b have a Pull_ID, which is driven or sent, for example, two cycles before the pull data.

Data functions are distributed amongst the programming engines 16. Connectivity to the DRAM memory 14a and the SRAM memory 14b is performed via command requests. A command request can be a memory request. For example, a command request can move data from a register located in the programming engine 16a to a shared resource, e.g., the DRAM memory 14a, SRAM memory 14b. The commands or requests are sent out to each of the functional units and the shared resources. Commands such as I/O commands (e.g., SRAM read, SRAM write, DRAM read, DRAM write, load data from a receive memory buffer, move data to a transmit memory buffer) specify either context relative source or destination registers in the programming engines 16.

In general, the data transfers between programming engines and memory resources designate the memory resource for pushing the data to a processing agent via the push bus having a plurality of sources that arbitrate use of the push bus, and designate the memory resource for receiving the data from the processing agent via the pull bus having a plurality of destinations that arbitrate use of the pull bus.

Read Process:

Referring to FIG. 3, a data read process 50 is executed during a read phase of the programming engines 16 by the push

buses 26a-26b. As part of the read process 50 the programming engine executes (52) a context. The programming engine 16 issues (54) a read command to the memory controllers 18a-18b, and the memory controllers 18a-18b processes (56) the request for one of the memory resources, i.e., the DRAM memory 14a or the SRAM memory 14b. For read commands, after the read command is issued (54), the programming engines 16 check (58) if the read data is required to continue the program context. If the read data is required to continue the program context or thread, the context is swapped out (60). The programming engine 16 checks (62) to ensure that the memory controllers 18a-18b have finished the request. When the memory controllers have finished the request, the context is swapped back in (64).

If the request is not required to continue the execution of the context, the programming engine 16 checks (68) if the memory controllers 18a-18b have finished the request. If the memory controllers 18a-18b have not finished the request, a loop back occurs and further checks (58) take place. If the memory controllers 18a-18b have finished the request, when the read data has been acquired from the memory resources, the memory controllers 18a-18b push (70) the data into the context relative input transfer register specified by the read command. The memory controller sets a signal in the programming engine 16 that

enables the context that issued the read to become active. The programming engine 16 reads (72) the requested data in the input transfer register and continues (74) the execution of the context.

Write Process:

Referring to FIG. 4, a data write process 80 is executed during a write phase of the programming engines 16 by the pull buses 28a-28b. During the write process 80 the programming engine executes (82) a context. The programming engine 16 loads (84) the data into the output transfer register and issues (86) a write command or request to the memory controllers 18a-18b. The output transfer register is set (88) to a read-only state. For write commands from the programming engines 16, after the output transfer register is set (88) to a read-only state, the programming engine 16 checks (90) if the request is required to continue the program context or thread. If yes, the context is swapped out (92).

If the write request is not required to continue the program context or thread, the memory controllers 18a-18b extracts or pulls (94) the data from the output transfer registers and signals (96) to the programming engines 16 to unlock the output transfer registers. The programming engine 16 then checks (98)

if the context was swapped out. If so, the context is swapped back (100) and if not, the programming engine 16 continues (102) the execution of the context. Thus, the signaled context can reuse the output transfer registers. The signal may also be used to enable the context to go active if it swapped out (100) on the write command.

Data Push Operation:

Referring to FIG. 5, a data push operation 110 that occurs in the push buses 26a-26b of the computer processing system 10, is shown in different processing cycles, e.g., cycle 0 through cycle 5. Each target, e.g., the DRAM memory 14a or the SRAM memory 14b, sends or drives (112) a Target_#_Push_ID to the push arbiters where the # indicates the number of different contexts such as context #0 through context #7. The Target_#_Push_ID is derived from the read command and a data error bit (e.g., the numbers following the target represent the source address incrementing in the Push_ID) for information it would like to push to the push arbiters 32a-32b. For Push_IDs, each letter indicates a push operation to a particular destination. A Push_ID destination of "none" indicates that the Push_ID is null. The target also sends the Target_#_Push_Data to the Push Arbiter.

The Push_ID and Push_Data are registered (114) and enqueued (116) into first-in, first-outs (FIFOs) in the push arbiters 32a-32b unless the Target_#_Push_Q_Full signal is asserted. This signal indicates that the Push_ID and Push_Data FIFOs for that specific target are almost full in the push arbiters 32a-32b. In this case, the push arbiters 32a-32b have not registered a Push_ID or Push_Data and the target does not change it. The channel changes the Push_ID and Push_Data that is taken by the push arbiters 32a-32b to those for the next word transfer or to null if it has no other valid transfer. Due to latency in the Push_Q_Full signal, the push arbiters 32a-32b should accommodate the worst case number of in-flight Push_IDs and Push_Data per target.

The push arbiters 32a-32b will arbitrate (118) every cycle between all valid Push_IDs and send intermediate Push_ID. The arbitration policy can be round robin, a priority scheme or even programmable. Multiple pushes of data from the push arbiters 32a-32b to the destination are not guaranteed to be in consecutive cycles. The push arbiters 32a-32b send (12) intermediate Push_Data and Push_ID is forwarded (120) to the destination. It is up to the target to update the destination address of each Push_ID it issues for each word of data it wishes to push. The Push_Data is forwarded (122) to the destination.

At the destination, the time from the destination getting the Push_ID to the destination getting Push_Data is fixed by one processing cycle.

Data Pull Operation:

Referring to FIG. 6, a data pull operation 130 that occurs in the pull buses 28a-28b of the computer processing system 10, is shown in different processing cycles (e.g., cycle 0 through cycle 7). Each target, e.g., the DRAM memory 14a or the SRAM memory 14b, sends or drives (132) the full Target_#_Pull_ID (i.e., the numbers following the target represents the source address incrementing in the Pull_ID) and length (derived from the write command) for information it would like to pull to the target. For Pull_IDs, each letter indicates a pull operation from a particular source, e.g., the memory resource 14. A Pull_ID source of "none" indicates that the Pull_ID is null. The target must have buffer space available for the pull data when it asserts its Pull_ID.

The Pull_ID is registered (134) and enqueued (136) into fist-in, first-outs (FIFO) in the pull arbiters 30a-30b, unless the Target_#_Pull_Q_Full signal is asserted. This signal indicates that the Pull_ID queue for that specific target is almost full in the pull arbiters 30a-30b. In this case, the pull

arbiters 30a-30b have not registered the Pull_ID and the target does not change it. The target changes a Pull_ID that is taken by the pull arbiters 30a-30b to that for the next burst transfer or to null if it has no other valid Pull_ID. Due to latency in the Pull_Q_Full signal, the pull arbiters 30a-30b should accommodate the worst case number of in-flight Pull_IDs per target.

The pull arbiters 30a-30b arbitrate (138) every cycle among the currently valid Pull_IDs. The arbitration policy can be round robin, a priority scheme or even programmable.

The pull arbiters 30a-30b forwards (140) the selected Pull_ID to the source. The time from the pull arbiters 30a-30b sending the Pull_ID to the source providing data is fixed in three processing cycles. The pull arbiters 30a-30b update the "source address" field of the Pull_ID for each new data item. The Pull_Data is pulled (142) from the source and sent to the targets.

The pull arbiters 30a-30b also assert (146) a Target_#_Take_Data to the selected target. This signal is asserted for each cycle a valid word of data is sent to the target. However, the assertions are not guaranteed to be on consecutive processing cycles. The pull arbiters 30a-30b only assert at most one Target_#_Take_Data signal at a time.

For transfers between targets and masters with different bus widths, the pull arbiters 30a-30b are required to do the adjusting. For example, the DRAM controller 18b may accept eight bytes of data per processing cycle but the programming engine 16 may only deliver four bytes per cycle. In this case, the pull arbiters 30a-30b can be used to accept four bytes per processing cycle, merge and pack them into eight bytes, and send the data to the DRAM controller 18a.

Other Embodiments:

It is to be understood that while the example above has been described in conjunction with the detailed description thereof, the foregoing description is intended to illustrate and not limit the scope of the invention, which is defined by the scope of the appended claims. Other aspects, advantages, and modifications are within the scope of the following claims.